REMARKS

Claims 1-3, 5-7 and 9-20 are presently pending.

Objection to the Specification

The specification has been objected to because of the Appendix, comprising pages 23-42 of the application. The Appendix has been converted to CD-ROM format and a CD is submitted herewith, in duplicate.

Rejection of Claims under 35 U.S.C. § 103

The focus of this response is whether the references cited by the Examiner include "a data repository for facilitating synchronization of user information maintained among more than two data sets, said data repository storing user information that is a super-set of all user information for which any user desires synchronization support". The Examiner currently is combining two references to meet this limitation, U.S. Patent No. 5,684,990 issued to Boothby (hereinafter "Boothby") in view of U.S. Patent No. 5,706,509 issued to Man-Hak Tso (hereafter "Man-Hak Tso"). However, neither of the references include the super-set feature for more than two data sets, either explicitly or implicitly, so combining the references cannot meet the limitation.

The Examiner clearly agrees that, "Boothby does not explicitly disclose establishing a data repository for facilitating synchronization of user information maintained more than two data sets, said data repository storing user information that is a superset of all user information for which any user desires synchronization support; and receiving a request for synchronizing at least one data set." The Examiner makes no argument that Boothby implicitly includes a super-set data repository for more than two data sets. Therefore, Boothby (admittedly) does not supply the missing limitation.

Man-Hak Tso does not include a super-set repository either. The Examiner relies on passages from columns 4 (more than two data sets, synchronized pairwise) and 6 (receiving requests, to be applied in turn with every other data set) that are reproduced on the following page.

Although this synchronization process is illustrated with only two data sets to synchronize, namely exemplary data sets D0 and D1, the present invention can easily be extended to synchronize more than two data sets. For more than two data sets, synchronization can be applied to pairs of data sets until all sets are equivalent. For instance, given four data sets D1', D2', D3', and D4', each data set may be synchronized in turn with every other data set. That is, D1' is synchronized in turn with D2', D3', and D4', then D2' is synchronized with D1', D3' and D4', etc. A more efficient implementation would run the Change Detection Method outlined in this invention on each of the data sets, and then merge the Change Lists (CL1, CL2, CL3, CL4). Thus, the present invention's method and apparatus for a two way synchronization also provides synchronization among any number of data sets (i.e. files).

FIGS. 5a-5c are exemplary embodiments of a system block diagram with the implementation of the synchronization method and apparatus of the present invention. The present invention may be used to synchronize data between data sets D0 and D1, belonging to application app0 and application app1 respectively. A variety of configurations are possible. For example, D0 may reside in a satellite device (e.g. a notebook or a hand held computer, such as an Apple® Newton, a Sharp® Wizard, or a Caslo® BOSS) and D1 may reside on a host computer (e.g. a desktop or a notebook PC) as illustrated in FIG. 5a. Further, D0 and D1 may reside on the same system as illustrated in FIG. 5b. D0 and D1 may also reside on two different PC's linked by a computer network as illustrated in FIG. 5c. In addition, app0 and app1 may be the same application. The present invention may be implemented for synchronization of any two or more data sets and is not limited to the exemplary configurations illustrated herein.

Applicants find nothing in these passages which even arguably includes a super-set of all user information for which any user desires synchronization support. First, it isn't there. Second, it isn't a logical extension of Man-Hak Tso, because synchronization in Man-Hak Tso is expressly pairwise between individual data sets. Col. 4, lines 51-54. Man-Hak Tso teaches away from a central repository, preferring to merge change lists and apply the merged change list to pairwise synchronization in a manner that is suggested but not clearly disclosed. Col. 4, lines 56-59. Man-Hak Tso uses a pairwise synchronization or change list merging to avoid creation of a "data repository storing user information that is a super-set of all user information for which any user desires synchronization support". Indeed, Man-Hak Tso emphasizes that the disclosed synchronization reduces file size, for instance at col. 7, lines 61-64, which teaches away from creating a super-set data repository.

It should not be surprising that Man-Hak Tso lacks the claimed features and sophistication of this invention. Man-Hak Tso laments how primitive synchronization was in 1995, throughout columns 1-2. For instance, "The main synchronization technique available today [in 1995] is referred to as file synchronization. ... A typical implementation uses time stamps which a computer's file system attaches to each file to determine which files are now or have been modified. The older files are overwritten with the newer files by the same name." Col. 1, lines 27-33. The principal teaching of Man-Hak Tso is a way to synchronize using a change list (or merged change lists) on a record-by-record basis, across files from different applications, instead of using file-by-file updating. Man-Hak Tso further emphasizes the difference between synchronizing

instances of data managed by a single application and instances of data managed by different applications, at col. 2, lines 35-42. Man-Hak Tso does not use or suggest use of a data repository.

The Examiner cites a passage from column 15 of Man-Hak Tso to motivate combination of the references to meet the missing limitation. The passage reads:

> What has been described is a method and an apparatus for performing record level synchronization on two or more applications. Record level synchronization overcomes the limitations of the prior art technique by synchronizing the individual data items (records) in a file. It uses knowledge of

Applicants find nothing in this passage describing a super-set data repository. Record level synchronization, addressed by this passage, is found in Boothby, which is more sophisticated than the older Man-Hak Tso reference. This passage does nothing to motivate a "data repository storing user information that is a superset of all user information for which any user desires synchronization support".

As neither of the cited references include the claimed feature, neither would their combination. Nor does the passage from column 15, argued by the Examiner as motivating the combination, fill in what's missing.

Regarding all of the Section 103(a) rejections, the Examiner seems to have an outdated standard in mind for a *prima facie* case of obviousness. The MPEP in Sections 716.01(a) and 716.03(b) cites *In re Fielder* for propositions other than the proposition that the Examiner argues. What the MPEP now cites for the Examiner's burden of proving a *prima facie* case is *In re Lee*, not *In re Fielder*. It is fundamental, as indicated in MPEP Section 2143.01 (8th Ed. Rev. 2, June 2004), that the Examiner rely on some evidentiary quality suggestion to modify Boothby:

> Obviousness can only be established by combining or modifying the teachings of the prior art to produce the claimed invention where there is some teaching, suggestion, or motivation to do so found either explicitly or implicitly in the references themselves or in the knowledge generally available to one of ordinary skill in the art. "The test for an implicit showing is what the combined teachings, knowledge of one of ordinary skill in the art, and the nature of the problem to be solved as a whole would have suggested to those of ordinary skill in the art." *In re Kotzab*, 217 F.3d 1365, 1370, 55 USPQ2d 1313, 1317 (Fed. Cir. 2000). See also *In re Lee*, 277 F.3d 1338, 1342-44, 61 USPQ2d 1430, 1433-34 (Fed. Cir. 2002) (discussing the importance of relying on objective evidence and making specific factual findings with respect to the motivation to combine references); *In re Fine*, 837 F.2d 1071, 5 USPQ2d 1596 (Fed. Cir. 1988); *In re Jones*, 958 F.2d 347, 21 USPQ2d 1941 (Fed. Cir. 1992).

The latest two updates to this section of the MPEP cite *In re Lee*, in which the Federal Circuit clarified the need for evidentiary quality support of an Examiner's factual basis for finding a teaching, suggestion or motivation in the prior art (as opposed to the Examiner's opinion), 277 F.3d at 1343-44:

> As applied to the determination of patentability *vel non* when the issue is obviousness, "it is fundamental that rejections under 35 U.S.C. § 103 must be based on evidence comprehended by the language of that section." *In re Grasselli*, 713 F.2d 731, 739, 218 U.S.P.Q. (BNA) 769, 775 (Fed. Cir. 1983). ... "The factual inquiry whether to combine references must be thorough and searching." *Id.* It must be based on objective evidence of record. This precedent has been reinforced in myriad decisions, and cannot be dispensed with. [citation omitted] The need for specificity pervades this authority. *See, e.g., In re Kotzab*, 217 F.3d 1365, 1371, 55 U.S.P.Q.2D (BNA) 1313, 1317 (Fed. Cir. 2000) ("particular findings must be made as to the reason the skilled artisan, with no knowledge of the claimed invention, would have selected these components for combination in the manner claimed"); *In re Rouffet*, 149 F.3d 1350, 1359, 47 U.S.P.Q.2D (BNA) 1453, 1459 (Fed. Cir. 1998) ("even when the level of skill in the art is high, the Board must identify specifically the principle, known to one of ordinary skill, that suggests the claimed combination. In other words, the Board must explain the reasons one of ordinary skill in the art would have been motivated to select the references and to combine them to render the claimed invention obvious."); *In re Fritch*, 972 F.2d 1260, 1265, 23 U.S.P.Q.2D (BNA) 1780, 1783 (Fed. Cir. 1992) (the examiner can satisfy the burden of showing obviousness of the combination "only by showing some objective teaching in the prior art or that knowledge generally available to one of ordinary skill in the art would lead that individual to combine the relevant teachings of the references"). ... In its decision on Lee's patent application, the Board rejected the need for "any specific hint or suggestion in a particular reference" to support the combination of the Nortrup and Thunderchopper references. Omission of a relevant factor required by precedent is both legal error and arbitrary agency action.

The point that Applicants have made and continue to rely on is that no evidentiary quality motivation has been supplied in any of the rejections in this case that would shift the burden to the Applicants to supply extrinsic evidence of non-obviousness. Citation of *In re Fielder* does not respond to Applicants' position.

With this analysis of Man-Hak Tso and the standard imposed by *In re Lee* in mind, we respectfully traverse the Examiner's arguments.[1]

Regarding claim 1, the present application, at page 5, provides support and description in the Summary of Invention for using a super-set data repository.

> The GUD introduces a third data set, a middleware database. This third data set provides a super-set of the other two client data sets. Therefore, if the user now includes a third client, such as a server computer storing user information, the synchronization system of the present invention has all the information necessary for synchronizing the new client, regardless of whether any of the other clients are currently available. The system can, therefore, correctly propagate information to any appropriate client without having to "go back" to (i.e., connect to) the original client from which that data originated.

The super-set data repository of claim 1 is nowhere to be found in Boothby or Man-Hak Tso. Nor do the references teach or enable a data repository of user information for which any user desires synchronization support. Pairwise synchronization is not equivalent to using a superset database, as explained in the application, because pairwise synchronization requires immediate availability of the source dataset clients.

In the sections that follow, we generally follow the order in which the Examiner addressed the dependent claims, except where noted.

The Examiner treats **claims 2 and 16** collectively, but we focus on claim 2, because it propagates data to a different place than claim 16. Claim 2 addresses propagating data to the data repository. Boothby col. 3, lines 46-50 does not teach updating a data repository "storing user information that is a super-set of all user information for which any user desires synchronization support". Boothby teaches separate status files for each pair of live data sets and ordered synchronization using the multiple status files, which would require availability of multiple source dataset clients. This teaches away from claim 2. This is an additional reason why claim 2 should be allowable over the cited references. Claim 16 should be allowable for at least the same reasons as claim 1, from which it depends.

---

[1] Applicants respectfully request a direct response to the positions previously submitted, that (1) Boothby does not teach or suggest modifying the status file P to be a super-set of information from more than two data sets to which users desire synchronization; (2) Boothby provides objective evidence of non-obviousness (teaching away), because he approaches synchronization among more than two data sets in a different way, pair-wise instead of using a super-set data repository; and (3) modifying Boothby in the manner that the Examiner proposes would impermissibly change the principle of operation described by Boothby.

For this response, Applicants will respond that **claims 3 and 17** are allowable for the same reasons as claim 1, from which they depend.

The Examiner next addresses claim 12, which depends from claim 11, so we address **claims 11 and 12** together, focusing on claim 11. To understand mapping and what Boothby does, we begin with the Official Action, page 4, where the Examiner cites Boothby col. 6, lines 40-49 and col. 7, lines 12-41, as "storing at least one mapping". These passages discuss the temporary synchronization workspace S. See, col. 5, lines 14-15 ("The synchronization workspace S is a temporary memory workspace used by the synchronization program."). The temporary synchronization workspace is not stored or otherwise persisted. Col. 4, lines 12-14 ("FIG. 2 shows a handheld database, status file, desktop database, and a temporary workspace ..."). What Boothby persists is the status file P. See, col. 5, lines 9-13 ("abbreviations: P for status file, N for handheld computer database, V for desktop computer database, and S for synchronization workspace"); col. 5, lines 45-51 ("The status file P, which is saved after a synchronization and used as input to the next synchronization, is a file containing one record per pair of synchronized handheld and desktop records. Each status file ... is identified by only one set of key fields or IDs.") The formats of Boothby's P, N, V and S files are depicted in Figure 2.

Turning to claims 11 and 12, the status file P that Boothby saves, stores or persists does not include first and second identifiers. Figure 2 makes clear that status file P does not include first and second identifiers and col. 5, lines 45-51 ("Each status file ... is identified by only one set of key fields or IDs") reinforces what is clear in the figure. The temporary workspace is where files are matched, using on-the-fly matching that Boothby refers to but does not describe. Col. 5, lines 6-9 ("The following descriptions assume that all of the corresponding records of the handheld database and the desktop database have already been mapped using such existing methods.") Again, the temporary workspace is not persisted, only the status file P. Also, Boothby emphasizes steps that minimize the size of the stored status file P. See, col. 8, lines 49-51. So Boothby teaches away from a file that persists first and second identifiers, preferring on-the-fly matching and minimized storage requirements for pairwise synchronization. Boothby's lack of a persistent mapping that includes at least a first and

second identifier is a further reason that claims 11-12 should be allowable over the cited
references.

The Examiner addresses **claims 5 and 6**, arguing that Boothby discloses a
"grand unification database". Grand unification database is not a term commonly used
in the art, so we consult the Summary of Invention for its meaning.

> The present invention introduces the notion of a reference database: the
> Grand Unification Database or GUD. By storing the data that is actually
> being synchronized (i.e., storing the actual physical body of a memo, for
> instance) inside an extra database (or by specially-designated one of the
> client data sets) under control of a central or core synchronization engine,
> rather than transferring such data on a point to point basis, the system of the
> present invention provides a repository of information that is available at all
> times and does not require that any other synchronization client (e.g., PIM
> client or hand held device) be connected. Suppose, for instance, that a user
> has two synchronization clients: a first data set residing on a desktop
> computer and a second data set residing on a hand held device. The GUD
> introduces a third data set, a middleware database. This third data set
> provides a super-set of the other two client data sets.

Applicants have reviewed col. 3, lines 15-23 and do not find GUD or anything GUD-like.
This is an additional reason that claims 5 and 6 (and **claim 7** which depends from 6)
should be allowable over the cited references.

The Examiner rejects **claims 9 and 18** "in the analysis of claim 1" without citing
any passages or providing any analysis, other than reference to claim 1. We focus on
claim 9, which adds to claim 1, "each data set comprises a plurality of data records, and
wherein each data record is represented within the data repository." In Boothby, only
the temporary workspace merges the contents of two data sets. As the temporary
workspace is not persisted, it cannot serve as a data repository. This is an additional
reason for claim 9 to be allowable over the cited art. For this response, Applicants
respond that claim 18 is allowable for at least the same reasons as claim 1.

The Examiner next rejects **claims 10-11**, which we have addressed along with
claim 12.

Claims 13-15 depend on claim 11 and, in turn claim 1. Claims 13-15 should be
allowable for at least the same reasons as claims 1 and 11.

Regarding **claim 19**, the Examiner cites col. 6, lines 19-31, for the limitation,
"wherein user information is stored at the data repository as unformatted blob data."

The cited passage discusses generating a To-Do list and using a decision matrix. Applicants do not understand what this has to do with unformatted blob data.

Regarding **claim 20**, cols. 7 and 8, lines 26-67 and 1-51, for the method further including, "providing at least one type module for facilitating interpretation of user information stored as unformatted blob data at the data repository." The cited passage elaborates on generating a To-Do list and using a decision matrix. Applicants do not understand what this has to do with unformatted blob data.

That the passages cited regarding claims 19-20 do not have anything to do with blob data is a further reason why claims 19-20 should be allowable over the cited references.
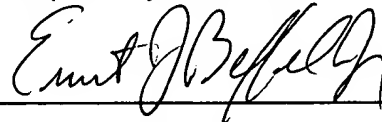
CONCLUSION

Applicants respectfully submit that the claims, as stated and amended herein, are in condition for allowance and solicit acceptance of the claims, in light of these remarks. If the Examiner disagrees and sees amendments that might facilitate allowance of the claims, a call would be appreciated.

Should any questions arise, the undersigned can ordinarily be reached at his office at 650-712-0340 from 8:30 to 5:30 PST, M-F and can be reached at his cell phone 415-902-6112 most other times.

Respectfully submitted,

Dated: 28 June 2004

Ernest J. Beffel, Jr., Reg. No. 43,489

HAYNES BEFFEL & WOLFELD LLP
P.O. Box 366
Half Moon Bay, CA 94019
(650) 712-0340 phone
(650) 712-0263 fax

5   <u>DATA PROCESSING ENVIRONMENT WITH METHODS PROVIDING</u>
         <u>CONTEMPORANEOUS SYNCHRONIZATION OF TWO OR MORE CLIENTS</u> **RECEIVED**

10   RELATED APPLICATIONS

[0001]   The present application is a continuation of co-pending U.S. Application No.

09/136,212, filed August 18, 1998, <u>now U.S. Patent 6,275,831;</u> which is related to and claims

the benefit of priority from the following commonly owned, formerly co-pending U.S.

provisional patent applications: serial no. 60/069,731, filed December 16, 1997, and entitled

15   DATA PROCESSING ENVIRONMENT WITH SYNCHRONIZATION METHODS

EMPLOYING A UNIFICATION DATABASE; serial no. 60/094,972, filed July 31, 1998, and

entitled SYSTEM AND METHODS FOR SYNCHRONIZING TWO OR MORE DATASETS;

and serial no. 60/094,824, filed July 31, 1998, and entitled DATA PROCESS ENVIRONMENT

WITH METHODS PROVIDING CONTEMPORANEOUS SYNCHRONIZATION OF TWO

20   OR MORE CLIENTS. The disclosures of the foregoing provisional applications are hereby

incorporated by reference in their entirety, including any appendices or attachments thereof, for

all purposes. The present application is also related to the following co-pending, commonly

owned U.S. patent application, the disclosures of which are hereby incorporated by reference in

their entirety, including any appendices or attachments thereof, for all purposes: serial no.

25   09/136,215 ~~(Attorney Docket No. SF/0018.03)~~, filed August 18, 1998, <u>now U.S. Patent</u>

<u>6,295,541,</u> and entitled SYSTEM AND METHODS FOR SYNCHRONIZING TWO OR MORE

DATASETS. The present application is also related to the following commonly owned U.S.

patent applications, the disclosures of which are hereby incorporated by reference in their

entirety, including any appendices or attachments thereof, for all purposes: serial no. 08/609,983,

30   filed February 29, 1996, <u>now U.S. Patent 5,845,257,</u> and entitled SYSTEM AND METHODS

FOR SCHEDULING AND TRACKING EVENTS ACROSS MULTIPLE TIME ZONES,~~ now~~

~~U.S. patent no. 5,845,257~~; serial no. 09/020,047, filed February 6, 1998, <u>now U.S. Patent</u>

<u>6,216,131,</u> and entitled METHODS FOR MAPPING DATA FIELDS FROM ONE DATA SET

TO ANOTHER IN A DATA PROCESSING ENVIRONMENT, ~~now U.S. patent no. 6,216,131,~~ and serial no. 08/923,612, filed September 4, 1997, and entitled SYSTEM AND METHODS FOR SYNCHRONIZING INFORMATION AMONG DISPARATE DATASETS.

5 **COMPUTER PROGRAM LISTING APPENDIX**

[0002] The file of this patent contains a computer program listing appendix submitted on one compact disc, including a duplicate compact disc, in a file named "APPENDIX.TXT", having a date of creation of June 28, 2004 and a size of 28,672 bytes. The contents of the compact disc are hereby incorporated by reference.

10

COPYRIGHT NOTICE

[0003] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile 15 reproduction by anyone of the patent document or the patent disclosure as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

20 [0004] The present invention relates generally to management of information or sets of data (i.e., "data sets") stored on electronic devices and, more particularly, to a system implementing methods for maintaining synchronization of disparate data sets among a variety of such devices, particularly synchronizing three or more devices at a time.

[0005] With each passing day, there is ever increasing interest in providing 25 synchronization solutions for connected information appliances. Here, the general environment includes "appliances" in the form of electronic devices such as cellular phones, pagers, hand-held devices (e.g., PalmPilot™ and Windows™ CE devices), as well as desktop computers and the emerging "NC" device (i.e., a "network computer" running, for example, a Java virtual machine or a browser).

[0006]   As the use of information appliances is ever growing, often users will have their data in more than one device, or in more than one desktop application. Consider, for instance, a user who has his or her appointments on a desktop PC (personal computer) but also has a battery-powered, hand-held device for use in the field. What the user really wants is for the information of each device to remain synchronized with all other devices in a convenient, transparent manner. Still further, the desktop PC is typically connected to a server computer, which stores information for the user. The user would of course like the information on the server computer to participate in the synchronization, so that the server also remains synchronized.

[0007]   A particular problem exists as to how one integrates disparate information -- such as calendaring, scheduling, and contact information -- among multiple devices, especially three or more devices. For example, a user might have a PalmPilot ("Pilot") device, a REX™ device, and a desktop application (e.g., Starfish Sidekick running on a desktop computer). Currently, in order to have all three synchronized, the user must follow a multi-step process. For instance, the user might first synchronize data from the REX™ device to the desktop application, followed by synchronizing data from the desktop application to the Pilot device. The user is not yet done, however. The user must synchronize the Pilot back to the REX™ device, to complete the loop. Description of the design and operation of the REX™ device itself (available as Model REX-3, from Franklin Electronic Publishers of Burlington, NJ) is provided in commonly-owned U.S. patent application serial no. 08/905,463, filed August 4, 1997, and entitled, USER INTERFACE METHODOLOGY FOR MICROPROCESSOR DEVICE HAVING LIMITED USER INPUT, the disclosure of which is hereby incorporated by reference.

[0008]   Expectantly, the above point-to-point approach is disadvantageous. First, the approach requires user participation in multiple steps. This is not only time consuming but also error prone. Further, the user is required to purchase at least two products. Existing solutions today are tailored around a device-to-desktop PIM (Personal Information Manager) synchronization, with no product capable of supporting concurrent synchronization of three or more devices. Thus for a user having three or more devices, he or she must purchase two or more separate synchronization products. In essence, existing products to date only provide peer-to-peer synchronization between two points, such as between point A and point B. There is no product providing synchronization from, say, point A to point B to point C, all at the same time.

Instead, the user is required to perform the synchronization manually by synchronizing point A to point B, followed by synchronizing point B to point C, then followed by point C back to point A, for completing the loop.

[0009] As a related disadvantage, existing systems adopt what is, in essence, an approach having a "hard-coded" link for performing synchronization for a given type of data. Suppose, for example, that a user desires to update his or her synchronization system for now accommodating the synchronization of e-mail data (e.g., Microsoft® Outlook e-mail). With existing synchronization products, the user cannot simply plug in a new driver or module for supporting this new data type. To the point, existing products today do not provide a generic framework into which data type-specific modules may plug into. As a result, these products are inflexible. In the event that the user encounters a new type of data for which synchronization is desired, he or she is required to update all or substantially all of the synchronization product. The user cannot simply plug in a driver or module for supporting synchronization of the new data type. All told, existing synchronization products today assume that users will only perform point-to-point (i.e., two device) synchronization, such as between a hand-held device and a desktop application running on a PC.

[0010] This assumption is far removed from reality, however. Users are more likely today to have data among multiple devices, such as among a desktop computer, a server computer (e.g., company network at the user's place of employment), and two or more portable devices (e.g., a laptop computer and a hand-held device). Given the substantial effort required to manually keep three or more devices synchronized, the benefits of synchronization largely remain unrealized for most computer and information application users today.

[0011] What is needed is a system providing methods which allows a user of information processing devices to synchronize user information, such as user-supplied contact lists, from one device to any number of other devices, including three or more devices concurrently. The present invention fulfills this and other needs.

SUMMARY OF THE INVENTION

[0012] The present invention introduces the notion of a reference database: the Grand Unification Database or GUD. By storing the data that is actually being synchronized (i.e.,

storing the actual physical body of a memo, for instance) inside an extra database (or by specially-designated one of the client data sets) under control of a central or core synchronization engine, rather than transferring such data on a point-to-point basis, the system of the present invention provides a repository of information that is available at all times and does not require

5   that any other synchronization client (e.g., PIM client or hand-held device) be connected. Suppose, for instance, that a user has two synchronization clients: a first data set residing on a desktop computer and a second data set residing on a hand-held device. The GUD introduces a third data set, a middleware database. This third data set provides a super-set of the other two client data sets. Therefore, if the user now includes a third client, such as a server computer

10   storing user information, the synchronization system of the present invention has all the information necessary for synchronizing the new client, regardless of whether any of the other clients are currently available. The system can, therefore, correctly propagate information to any appropriate client without having to "go back" to (i.e., connect to) the original client from which that data originated.

15   **[0013]**   Internally, the system of the present invention employs "type plug-in" modules, each one for supporting a particular data type. Since the core synchronization engine treats data generically as "blob" objects, type-specific support is provided by the corresponding plug-in module. Each plug-in module is a type-specific module having an embedded record API (application programming interface) that each synchronization client may link to, for providing

20   type-specific interpretation of blob data. For instance, the system may include one type-specific record API for contact information, another for calendar information, and yet another for memo information. In this manner, each client may employ a type-specific API for correctly interpreting and processing particular blob data. The engine, on the other hand, is concerned with correct propagation of data, not interpretation of that data. It therefore treats the data itself

25   generically. In this fashion, the present invention provides a generic framework supporting concurrent synchronization of an arbitrary number of synchronization clients or devices.

       [0014]   Also internally, the synchronization system of the present invention employs an "action queue," for optimizing the actual synchronization work performed. In contrast to conventional point-to-point (i.e., binary) synchronization systems, the synchronization system of

30   the present invention does not immediately transmit updates or changes as soon as they are

detected. Instead, the system determines or tabulates changes, net of all clients, before undertaking the actual work (e.g., record insertion) of synchronizing a particular client. In particular, all actions or tasks which are to be performed for a client by the system during synchronization are queued in the outbound action queue. This allows the system to apply synchronization logic or intelligence to the queue for further improving system performance, such as eliminating any activities which are redundant or moot. For example, if the system receives a request from two different clients to update a given record (i.e., conflict), the system, applying internal synchronization logic, can eliminate propagating the first update, as it is rendered moot by the second update. In this manner, the system can apply a first-level resolution of requests that are conflicting (or complimentary) and, as a result, eliminate those synchronization activities which are redundant or moot.

[0015] An exemplary method for synchronizing multiple data sets includes first establishing a data repository for facilitating synchronization of user information maintained among multiple data sets, the data repository storing user information from the data sets. At least one mapping is stored which specifies how user information may be transformed for storage at a given data set. Upon receiving a request for synchronizing at least one data set, the system may, based on user information stored at the data set(s) and based on the mapping, propagate to the data repository from each data set(s) any changes made to the user information, to the extent that such changes can be reconciled with user information already present at the data repository. Further, based on user information stored at said data repository and based on the mapping, the system may propagate to each data set(s) any changes to the user information which have been propagated to the data repository, to the extent that such changes are not present at the data set.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] Fig. 1A is a block diagram of a computer system in which the present invention may be embodied.

[0017] Fig. 1B is a block diagram of a software system of the present invention for controlling operation of the system of Fig. 1A.

[0018] Fig. 2 is a block diagram of the synchronization system of the present invention.

[0019] Fig. 3 is a block diagram of a GUD of the present invention.

[0020]   Figs. 4A-C are flow charts of the operation of the synchronization system of the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

5   [0021]   The following description will focus on the presently-preferred embodiment of the present invention, which is operative in an environment typically including desktop computers, server computers, and portable computing devices, occasionally or permanently connected to one another, where synchronization support is desired. The present invention, however, is not limited to any particular environment or device. Instead, those skilled in the art will find that the

10   present invention may be advantageously applied to any environment or application where contemporaneous synchronization among an arbitrary number of devices (i.e., "synchronization clients"), especially three or more devices, is desirable. The description of the exemplary embodiments which follows is, therefore, for the purpose of illustration and not limitation.

15   **System hardware and software**

[0022]   The present invention may be embodied on an information processing system such as the system 100 of Fig. 1A, which comprises a central processor 101, a main memory 102, an input/output (I/O) controller 103, a keyboard 104, a pointing device 105 (e.g., mouse, pen device, or the like), a screen or display device 106, a mass storage 107 (e.g., hard disk, removable

20   floppy disk, optical disk, magneto-optical disk, flash memory, or the like), one or more optional output device(s) 108, and an interface 109. Although not shown separately, a real-time system clock is included with the system 100, in a conventional manner. The various components of the system 100 communicate through a system bus 110 or similar architecture. In addition, the system 100 may communicate with other devices through the interface or communication port

25   109, which may be an RS-232 serial port or the like. Devices which will be commonly connected to the interface 109 include a network 151 (e.g., LANs or the Internet), a laptop 152, a handheld organizer 154 (e.g., the REX™ organizer, available from Franklin Electronic Publishers of Burlington, NJ), a modem 153, and the like.

[0023]   In operation, program logic (implementing the methodology described below) is

30   loaded from the storage device or mass storage 107 into the main memory 102, for execution by

the processor 101. During operation of the program (logic), the user enters commands through the keyboard 104 and/or pointing device 105 which is typically a mouse, a track ball, or the like. The computer system displays text and/or graphic images and other data on the display device 106, such as a cathode-ray tube or an LCD display. A hard copy of the displayed information, or

5   other information within the system 100, may be obtained from the output device 108 (e.g., a printer). In a preferred embodiment, the computer system 100 includes an IBM PC-compatible personal computer (available from a variety of vendors, including IBM of Armonk, New York) running Windows 9x or Windows NT (available from Microsoft Corporation of Redmond, Washington). In a specific embodiment, the system 100 is an Internet or intranet or other type of

10   network server and receives input from and sends output to a remote user via the interface 109 according to standard techniques and protocols.

[0024]   Illustrated in Fig. 1B, a computer software system 120 is provided for directing the operation of the computer system 100. Software system 120, which is stored in system memory 102 and on storage (e.g., disk memory) 107, includes a kernel or operating system (OS)

15   140 and a windows shell 150. One or more application programs, such as client application software or "programs" 145 may be "loaded" (i.e., transferred from storage 107 into memory 102) for execution by the system 100.

[0025]   System 120 includes a user interface (UI) 160, preferably a Graphical User Interface (GUI), for receiving user commands and data and for producing output to the user.

20   These inputs, in turn, may be acted upon by the system 100 in accordance with instructions from operating system module 140, windows module 150, and/or client application module(s) 145. The UI 160 also serves to display the user prompts and results of operation from the OS 140, windows 150, and application(s) 145, whereupon the user may supply additional inputs or terminate the session. In the preferred embodiment, OS 140 and windows 150 together

25   comprise Microsoft Windows software (e.g., Windows 9x or Windows NT). Although shown conceptually as a separate module, the UI is typically provided by interaction of the application modules with the windows shell and the OS 140.

[0026]   Of particular interest herein is a synchronization system or "Synchronizer" 200 of the present invention, which implements methodology for contemporaneous synchronization of

30   an arbitrary number of devices or "clients." Before describing the detailed construction and

operation of the Synchronizer 200, it is helpful to first briefly review the basic application of synchronization to everyday computing tasks.

Brief overview of synchronization

5   **A. Introduction**

[0027]   Many software applications, such as personal productivity applications as Starfish Sidekick® and Lotus® Organizer, have sets of data or "data sets" (e.g., address books and calendars). Consider for instance a user scenario where an account executive needs to coordinate contacts and events with other employees of the XYZ corporation. When traveling, this

10   executive carries a laptop PC with Starfish Sidekick® installed. At home, she and her husband use Lotus® Organizer to plan their family's activities. When on family outings, the account executive carries her PalmPilot™ hand-held organizer. As the foregoing illustrates, a user often needs a means for synchronizing selected information from the data sets his or her applications rely upon. The account executive would not want to schedule a business meeting at the same

15   time as a family event, for example.

[0028]   Conventionally, the process of synchronizing or reconciling data sets has been a binary process -- that is, two logical data sets are synchronized at a time. Any arbitrary synchronization topology will be supported. Here, the system guarantees synchronization stability and the avoidance of undesirable side effects (cascading updates, record duplication, or

20   the like). Data sets do not need to be directly connected but, instead, can be connected via a "store-and-forward" transport, such as electronic mail.

               B. Synchronization design

               1. Synchronization type

25   [0029]   Data set synchronization may, for convenience of description, be divided into two types: content-oriented and record-oriented. Content-oriented synchronization correlates data set records based on the values of user-modifiable fields. Value correlation requires semantic (or at least advanced syntactic) processing that the human brain is very good at and computers are not. For example, a record in one data set with a name field valued "Johann S. Bach" and a record in a

30   second data set with a name field valued "J. S. Bach" could possibly refer to the same real-world

person. A human being might arrive at this conclusion by correlating associated data (addresses) or drawing upon external information (e.g., Bach is an unusual name in the U.S.). Creating program logic or code with the ability to make these type of decisions is computationally very expensive.

[0030] Record-oriented synchronization correlates data set records by assuming that each record can be uniquely identified throughout its lifetime. This unique identifier is usually implemented as a non-modifiable, hidden field containing a "Record ID". Record-oriented synchronization algorithms usually require maintaining a mapping from one set of record IDs to another. In a preferred embodiment, the system employs record-oriented synchronization.

[0031] Record-oriented synchronization is conceptually simple and may be summarized as follows. In the rules below, A and B refer to two data sets which have a synchronization relationship. The rules are assumed to be symmetrical.

1. A and B must track similar types of data (e.g., if A is an address book, then B must be an address book).
2. A record entered in A, will create a record in B.
3. A record modified in A, will modify the corresponding record in B.
4. If record A1 has been modified in A and the corresponding record B1 has been modified in B, the record with the latest timestamp takes precedence.

The rules presented above reduce the occurrence of undesirable side effects with a network of synchronized data sets.

2. Timestamps

[0032] The actual synchronization logic in synchronization systems often needs to make processing decisions based on comparing the time at which past events occurred. For example, it is necessary to know if a record was modified before or after the last synchronization transaction. This requires recording the time of various events. A "timestamp" value may be employed to this purpose. Typically, data sets involved in synchronization support timestamps, or can be supplied with suitable timestamps, in a conventional manner. In conjunction with the usage of timestamps to compare the relative timing of record creation or modification, the clocks on the respective devices may themselves be synchronized.

3. Record Transformations

[0033]   During synchronization, a synchronization system will typically transform records from one application-usage-schema set to another application-usage-schema set, such as transforming from a Starfish Sidekick® card file for business contacts to a corresponding

5   PalmPilot™ data set. Typically, there is a one-to-one relationship between records in these two data sets, that is, between the source and target data sets. If this is not the case, however, the component of the system that interacts with a non-conforming data set may include logic to handle this non-conformance.

[0034]   The record transformations themselves are a combination of field mappings and

10   conversions from a source record to a target record. Exemplary types of field mappings include, for instance, the following.

1. *Null*	Source field has no equivalent field in the target data set and is ignored during synchronization.
15	2. *One-to-One*	Map exactly one field in the target to one field in the source.
	3. *One-to-Many*	Map one field in the target to many fields in the source, such as parse a single address line to fields for number, direction, street, suite/apartment, or the like.
	4. *Many-to-One*	Map several fields in the target to one field in the source, such as
20		reverse the address line mapping above.

Similarly, exemplary field conversions may be defined as follows.

	1. *Size*	Source field may be larger or smaller in size than the target field.
25	2. *Type*	Data types may be different, such as float/integer, character vs. numeric dates, or the like.
	3. *Discrete Values*	A field's values may be limited to a known set. These sets may be different from target to source and may be user defined.

30   It is often the case that there are significant differences in the number, size, type and usage of fields between two data sets in a synchronization relationship. The specification of transformations is typically user-configurable, with the underlying system providing defaults.

[0035]   With an understanding of the basic process of synchronizing information or computing devices, the reader may now better appreciate the teachings of the present invention

35   for providing improved methodology for contemporaneous synchronization of an arbitrary

number of devices (i.e., synchronization clients). The following description focuses on specific modifications to a synchronization system for implementing the improved synchronization methodology.

5   **Synchronization system providing contemporaneous synchronization of two or more clients**
A. General design considerations

[0036]   The present invention introduces the notion of a "Grand Unification Database" (GUD) -- a central repository or reference database for user data. By storing the data that is actually being synchronized (i.e., storing the actual physical body of a memo, for instance) inside

10   an extra database (or by specially-designated one of the client data sets) under control of a central or core synchronization engine, rather than transferring such data on a point-to-point basis, the system of the present invention provides a repository of information that is available at all times and does not require that any other synchronization client (e.g., PIM client or hand-held device) be connected. Suppose, for instance, that a user has two synchronization clients: a first data set

15   residing on a desktop computer and a second data set residing on a hand-held device. The GUD introduces a third data set, a middleware database. This third data set provides a super-set of the other two client data sets. Therefore, if the user now includes a third client, such as a server computer storing user information (or other information which the user desires synchronization to), the synchronization system of the present invention has all the information necessary for

20   synchronizing the new client, regardless of whether any of the other clients are currently available. The system can, therefore, correctly propagate information to any appropriate client without having to "go back" to (i.e., connect to) the original client from which that data originated.

[0037]   Internally, the system of the present invention employs a driver-based architecture

25   providing type-specific "plug-in" modules, each one for supporting a particular data type. Since the core synchronization engine treats data generically as "blob" objects, type-specific support is provided by the corresponding plug-in module. Each plug-in module is a type-specific module having an embedded record API (application programming interface) that each synchronization client may link to, for providing type-specific interpretation of blob data. For instance, the

30   system may include one type-specific record API for contact information, another for calendar

information, and yet another for memo information. In this manner, each client may employ a type-specific API for correctly interpreting and processing particular blob data. The engine, on the other hand, is concerned with correct propagation of data, not interpretation of that data. It therefore treats the data itself generically. In this fashion, the present invention provides a

5   generic framework supporting concurrent synchronization of an arbitrary number of synchronization clients or devices.

[0038]   Also internally, the synchronization system of the present invention employs an "action queue," for optimizing the actual synchronization work performed. In contrast to conventional point-to-point (i.e., binary) synchronization systems, the synchronization system of

10   the present invention does not immediately transmit updates or changes as soon as they are detected. Instead, the system determines or tabulates changes, net of all clients, before undertaking the actual work (e.g., record insertion) of synchronizing a particular client. In particular, all actions or tasks which are to be performed for a client by the system during synchronization are queued in the outbound action queue. This allows the system to apply

15   synchronization logic or intelligence to the queue for further improving system performance, such as eliminating any activities which are redundant or moot. For example, if the system receives a request from two different clients to update a given record (i.e., conflict), the system, applying internal synchronization logic, can eliminate propagating the first update, as it is rendered moot by the second update. In this manner, the system can apply a first-level resolution

20   of requests that are conflicting or complementary and, as a result, eliminate those synchronization activities which are redundant or moot.

B. Overview of synchronization system internal architecture

[0039]   Fig. 2 is a block diagram illustrating a modular or high-level view of the

25   synchronization system 200. As shown, the synchronization system 200 includes a synchronization engine (core) 230 that is connected to both a Grand Unification Database(s) (GUD(s)) 210 and to an action queue 240. As also shown, the engine presents two interfaces, a client API 220 and type API 250, for communicating with components outside the core engine.

[0040]   The GUD 210, as previously described, serves as a central repository storing

30   record data and mappings which dictate how records are transformed (i.e., from one data set to

another). The synchronization engine 230 includes generic logic for managing the GUD 210, including locating and interpreting information in the GUD. Based on the information in the GUD 210 and client requests, the synchronization engine 230 builds the action queue 240, adding or removing specific tasks from the queue as necessary for carrying out synchronization

5   transactions. The action queue 240 itself is an array of task entries; it may grow or shrink, depending on the current number of entries that it stores. In the currently-preferred embodiment, the array is sorted by record ID, that is, according to the record ID of the corresponding record from the GUD. Since entries are sorted by record ID, the task of identifying entries in conflict is simplified.

10   **[0041]**   To communicate with the clients, the synchronization engine 230 employs the client API 220. The client API provides database engine-like functionality. For example, API function calls are provided for moving to records, reading records, and writing records. In the currently-preferred embodiment, clients accessors 221, 223 are "accessor" portions of the synchronization system which, in turn, communicate directly with the "real" clients, such as

15   REX. By implementing its architecture such that all clients communicate commonly through the client API 220, the system 200 provides plug-in capability for supporting new clients.

[0042]   In order for the system to correctly determine record information in the GUD 210, the synchronization engine 230 communicates with type drivers or modules (e.g., X type 251 and Y type 253) through the type API 250. As previously described, each type, such as calendar,

20   contacts, and the like, is associated with a particular type module. The type API 250 allows the synchronization engine 230 to ask common questions about information stored in the GUD 210. For example, if the synchronization engine 230 needs to determine whether two records are identical, it can request a record comparison operation by the corresponding type module, using the type API 250. In comparison to the client API 220, the type API 250 is comparatively small.

25   By implementing its architecture such that all type-specific requests are communicated commonly through the type API 250, the system 200 provides built-in extensibility. When support is desired for a new type, one need only plug in a new type module. Any client which wants to communicate with that new type now has automatically gained support for that new type. In the currently-preferred embodiment, a type module is unaware of any specific clients

which it supports. Clients, on the other hand, typically know what types that each desires to synchronize with.

[0043]   As also shown, each client accessor can communicate directly with the type modules, using a record API 260. In the currently-preferred embodiment, each type module

5   surfaces its own record API, such as record API 260 for type module 251. The underlying record API is specific for each type. Each accessor communicates with a desired type module, not through the synchronization engine 230, but instead through the exposed record API for the desired type. Thus, in effect, there is a direct communication path between client accessors and type modules. In typical use, the record API is employed by a client accessor to create or write

10   record-specific information. For example, if the client desires to write a "subject" for a contact record, the client, operating through the corresponding client accessor, can invoke the corresponding record API for requesting this service. In response to invocation of the record API, the corresponding type module would service the API call for assisting with creating or editing the underlying record, in the matter requested by the client. The actual work of creating

15   or editing the record is typically performed by the client; however, the corresponding type module returns specific information about the given type, so that the client knows exactly how the record is structured. As a simple example, the record API might return information indicating that a particular record type consists of a structure having four string data members, each being 64 bytes long. Based on such information, the client now knows how to interpret and

20   process that type.

C. Synchronization system detailed internal architecture
1. GUD

[0044]   Fig. 3 is a block diagram illustrating organization of a GUD 300. In the currently-

25   preferred embodiment, the system implements one GUD per type. For instance, if one were synchronizing contacts, calendars, and "to do"s (i.e., task-oriented information), one would have three GUDs, one for each type. As shown, each GUD database internally stores two sets of tables: mapping tables 320 and data table 310. The data table 310 stores the actual record data 313 (i.e., blob data), together with a unique reference (ref) ID or "GUD ID" 311. In the

30   presently-preferred embodiment, each reference ID (e.g., a 32-bit or 64-bit ID) is unique not only

within its particular GUD database but also across all GUD databases. Thus, for example, the system would not duplicate a calendar reference ID in the contact GUD database. With this approach, the individual data items are uniquely identified across the entire system. If desired, the GUD itself (or its data record portion) may be implemented as one of the actual client data

5 sets (i.e., one of the data sets serves as the GUD, or portion thereof).

[0045] Also shown, mapping tables 320 store entries comprising a reference ID 321, a source ID 322, a checksum or integrity value (e.g., CRC) 323, and a last modification (mod) timestamp 324. The reference ID 321 is the same ID as associated with a record in the data table 310. The source ID 322 is the record ID for the record, as it was received from the client. The

10 last modification timestamp 324 establishes when the record was last synchronized through the system. The timestamp (e.g., system time structure) reflects the time on the system clock of the machine which is being synchronized. Optionally, the system stores a comparison value or checksum (e.g., cyclic redundancy checking or CRC) 323, for use with those clients that do not support timestamps. If the checksum is not used, the system stores 0 as its value.

15 [0046] Each table itself is linked to a particular client, through a table ID, with the correspondence being stored as configuration information (which in the currently-preferred environment exists as a higher level than the synchronization engine). In this manner, each one of the mapping tables can be associated with an appropriate client. The end result is that the system maintains a mapping table for each client. Thus, for a given record ID, the system can

20 easily determine (from the above-described reference ID-to-source ID correspondence) where that record maps to for all clients. Consider, for instance, a particular record residing on a REX device. Based on the source ID for that record, the system can determine from the mapping table the corresponding mapping table item for that source ID. Now, the system has sufficient information allowing the particular record to be synchronized, as required by the user. When the

25 data is completely synchronized with all clients, all mapping tables in the system will store that record ID (i.e., the record ID is now common to all tables once the data is completely synchronized with all clients).

30

2. Action queue

[0047]   The action queue stores entries of a particular action type, which are used during synchronization to indicate all actions needed to be performed by the system. In the currently-preferred embodiment, six action types are defined:

5

      (1) GUD_UPDATE

      (2) GUD_ADD

      (3) GUD_DELETE

      (4) CLIENT_UPDATE

10   (5) CLIENT_ADD

      (6) CLIENT_DELETE

The first three action types or "GUD action types" indicate actions to be performed against the GUD. For example, if the system receives a new record from a client, it must add the new record

15   to the (corresponding) GUD; this is indicated by an action queue entry having a type of GUD_ADD. In operation, the system will not only add the record to the corresponding GUD but, also, will eventually add that record to other clients which are associated with that record as well (unless the user instructs otherwise). In a similar manner, a GUD_UPDATE action item or command will result in the system updating the corresponding GUD for a given record (e.g., as a

20   result of that record having been modified at the client), and a GUD_DELETE action item or command will result in the system deleting the record from the corresponding GUD (e.g., as a result of that record having been deleted at the client).

[0048]   The CLIENT action types are used to indicate particular synchronization work which is required to be performed for a particular client. Suppose, for instance, that the

25   synchronization engine determines that the REX client needs to be updated, as a result of actions undertaken by other clients; the REX client need not be currently available (e.g., need not be currently connected to the system). In such a case, the engine can post to the action queue appropriate action entries for indicating the synchronization work which is required to be performed the next time the REX client is connected. In a manner similar to that described

30   above for the GUD, the system can specify an update (CLIENT_UPDATE), add

(CLIENT_ADD), and/or delete (CLIENT_DELETE) action, on a per client basis. In the instance of an update or delete action, there already exists a corresponding mapping table item. For an add action, however, the system undertakes as its first action item the task of creating a new mapping table item. Therefore, when the add action is eventually performed, the table item will

5   be created as well. On the other hand, should the action be canceled, the mapping table item will not be created.

[0049]   Additional pieces of information are tracked by each entry in the action queue: (1) record data, (2) source client, and (3) timestamp. The record data is the actual data (or a reference to the actual data) obtained from the client. In this manner, the actual data may be

10   associated with a particular action. The source client indicates which client the action originated from. This is useful, for instance, during synchronization, so that the system does not attempt to synchronize the client from which the data just arrived. The timestamp stored in an action queue entry is the last modification time of the record from the source client. This is stored for possible use during conflict resolution (which is described in further detail below).

15   [0050]   As previously described, the entries in the action queue are sorted by reference ID. In this manner, the system can quickly determine action queue entries which are potentially in conflict. For example, if the queue contains three entries all having the same reference ID, the system must examine those entries for uncovering any conflicts. The actual conflict resolution rules applied in the system are described below.

20

3. Methodology of system operation

[0051]   Fig. 4A illustrates an overall methodology 400 of the present invention for providing synchronization contemporaneously among an arbitrary number of clients. At step 401, the system initializes all clients and types (data structures). At step 402, the system

25   establishes a loop for determining for each client what actions are to be performed. Here, the system begins building the action queue. Once the action queue or table has been built, the system proceeds to resolve any conflicts present. This is indicated by step 403. In particular at this step, the system performs housekeeping on the queue, removing any action entries which are unnecessary.

[0052]   Conflict resolution requires further explanation. As previously described, the entries in the action queue are sorted by reference ID. In this manner, the system can quickly determine action queue entries which are potentially in conflict. For example, if the queue contains three entries all having the same reference ID, the system must examine those entries for uncovering any conflicts. Not only are items in the action queue sorted by a reference ID but, as a second level of ordering, they are also sorted by action. GUD updates are always sorted to the top, thus establishing their priority over other types. Now, the following exemplary conflict resolution rules may be applied:

Rule 0:
                        GUD_UPDATE
        +   <u>< entry(ies) other than GUD_UPDATE></u>

                        GUD_UPDATE wins; delete all others

Rule 1:
                        GUD_UPDATE
        +   <u>GUD_UPDATE</u>

                        GUD_UPDATE with greatest timestamp wins (or display UI)

Rule 2:
                        GUD_UPDATE
        +   <u>GUD_DELETE</u>

                        GUD_UPDATE (take data over non-data)

Rule 3:
                        CLIENT_UPDATE
        +   <u>CLIENT_UPDATE (from another client)</u>

                        Leave both (i.e., same)

Once conflicts have been resolved the action queue is ready for use. Specifically, at step 404, the system processes all remaining action entries in the action queue. The actions themselves are performed on a transaction-level basis, where a transaction comprises all actions performed on a given record GUD ID. Thereafter, the system may perform cleanup, including closing any open databases and freeing any initialized data structures (e.g., type).

[0053]   Fig. 4B illustrates particular substeps which are performed in conjunction with step 402. The substeps are as follows. At step 421, the system determines all updates and adds originating from the client (i.e., the client currently being processed during the "for" loop). In essence, the system operates by asking the client for all modifications (e.g., updated or added

5   records) since last synchronization. Once these are learned, the system places them in the action queue, either as a GUD_UPDATE or GUD_ADD. If desired, a filter may be applied at this point, for filtering out any records which are desired to be omitted from the synchronization process. The next step, at step 422, is for the system to determine any deletions coming from the client. Note, here, that the update/add step (421) comes before the deletion determination step

10   (422). This allows the system to determine what is new before determining what has been deleted. As an optimization at this point, the system can look at the record count at the client for determining whether in fact there have been any deletions at all. In the event that the count indicates no deletions, the system can eliminate the time-consuming process of determining deletions (which may require the system to examine numerous records individually). At step

15   423, the system makes a reverse determination: determining any updates or adds which need to be sent from the GUD back to the client. The mapping table stores a timestamp indicating when the client was last synchronized as well as a timestamp for each record item. Accordingly, the system can determine whether the item needs to be updated or added at the client. In the currently-preferred embodiment, the timestamp is generated based on the system clock of the

20   client which is undergoing synchronization. Finally, at step 424, the system determines any deleted records in the GUD, for indicating which corresponding records should be deleted from the client. Specifically in the mapping table, each entry includes a deletion flag which may be set for indicating deletion of the corresponding record. These foregoing steps are performed for all clients undergoing synchronization, until the action queue is filled with the appropriate action

25   entries required for effecting synchronization.

[0054]   Fig. 4C illustrates particular substeps which are performed in conjunction with step 404. The substeps are as follows. At step 431, the system determines whether the action is from one client to another client. If the action is to a client, the system may simply proceed to update the client, as indicated by step 432. If, on the other hand, the action is from a client, the

30   system must update the GUD, as indicated at step 433, and, in turn, propagate the update to the

other clients, as indicated at step 434. The actual propagation is performed recursively invoking itself as client actions (rather than GUD actions). Here, the system fabricates a surrogate or fake action item which is then acted upon as if it were from the action queue. All the time during the method, the GUD has played an important role as a data source for those clients which are not currently available.

5

~~Appended herewith as an Appendix A are source code listings providing further description of the present invention.~~

[0055]   While the invention is described in some detail with specific reference to a single-

10   preferred embodiment and certain alternatives, there is no intent to limit the invention to that particular embodiment or those specific alternatives.

5

10   ~~Appendix A~~

```
//////////////////////////////////////////////////////////
// Synchronization hub.

void Synchronize( void )
{
        GetActions( );
        ResolveConflicts( );
        PerformActions( );

        return;
}


//////////////////////////////////////////////////////////
// Get the actions from the sources and GUD.

void GetActions ( )
{
        // Iterate through all of the managers and add their actions to the list.
        for ( TSObject* pObj = m_vecSources.First();
            pObj;
                pObj = m_vecSources.Next ( ) )
        {
                TSSource* pSource = (TSSource*) pObj;

                // Get the record map from the store for this manager.
                TSSourceManager*      pManager = pSource->SourceManager ( );
                TSRecordMap*      pMap    = pManager->RecordMap ( );
                TSStore*                pStore  = pMap->Store ( );

                // Get the number of items being operated on.
                TSUINT32 uSourceCount = pSource->Count ( );
                TSUINT32 uMapCount   = pMap->MapItemCount ( );

                // Filter the gud for this specific source.
                m_pStore->Filter ( pSource );

                // Get the last synchronization time for the source itself.
                TSDateTimeStamp& tsLastSync = pMap->LastSync ( );

                // Generate the source update actions.
                int iAddCount = GetActions_SourceUpdates ( pSource, pMap, tsLastSync );

                // Generate the source delete actions.
                GetActions_SourceDeletes ( pSource,
                                                    pMap,
                                                    tsLastSync,
                                                    ( (long)uSourceCount
                                                            -(long)uMapCount
                                                            -(long)iAddCount
                                                    ) != 0 );

                // Generate the GUD update actions.
                GetActions_GudUpdates ( pSource, pMap );

                // Generate the GUD delete actions.
```

```
                    GetActions_GudDeletes ( pSource, pMap );
              }

              // Remove the filtering which was put in place for a given source
5             m_pStore->Filter ( NULL );

              return;
}

10    //////////////////////////////////////////////////////////
      // Generate the source update actions.

      TSINT32 GetActions_SourceUpdates (
              TSSource*                    pSource,
15            TSRecordMap*         pMap,
              TSDateTimeStamp&     tsLastSync
              )
      {
              TSDateTimeStamp       dtsLastModification;
20
              // Filter the source based on the last synchronization time.  This
              // will ensure optimal performance for sources which can offer the
              // filter.
              pSource->Filter ( TSSOURCE_FILTER_MODIFICATIONS, pMap->LastModification ( ) );
25
              // Iterate through each record in the source and determine whether
              // or not the record has been modified since the last synchronization
              TSINT32 iAddCount = 0;

30            if ( pSource->MoveFirst ( ) )
              {
                    do
                    {
                            // Get the item to operate on.
35                          TSString                        strID   = pSource->ID ( );
                            TSRecordMapItem*        pItem   = pMap->CurrentMapItem (
      TSRECORDMAP_MAP_SOURCEID, (TSUINT32)(TSCSTR)strID );
                            TSRecordAction*            pAction = NULL;

40                          TSDateTimeStamp dtsSourceMod = pSource->LastModified ( );
                            TSUINT32        uCRC        = pSource->CRC ( );

                            // If the record exists in the map then this is an update
                            // not an add.
45                          if ( pItem )
                            {
                                    // If there was a CRC value returned from the source we should assume that
                                    // the source does not have last modification times on the record level and
                                    // we should compare the last known crc with the given one to determine
50                                  // modification.
                                    if ( uCRC != 0 )
                                    {
                                            if ( uCRC != pItem->CRC ( ) )
                                                    pAction = new TSRecordAction (
55    TSRECACTIONTYPE_GUD_UPDATE, pSource, pItem );
```

```
                                }
                            else
                            {
                                if ( dtsSourceMod > pMap->LastModification ( ) )
                                    pAction = new TSRecordAction (
TSRECACTIONTYPE_GUD_UPDATE, pSource, pItem );
                            }
                    }
                    // If the record did not exist in the record map it must be a new record.
                    // Therefore we can add a new gud record and create a map for it.
                    else
                    {
                        TSRecord* pRecord = m_pStore->CreateRecord ( );

                        pItem   = pMap->CreateMapItem ( pSource->ID ( ), pRecord );
                        pAction = new TSRecordAction ( TSRECACTIONTYPE_GUD_ADD,
pSource, pItem );

                        iAddCount++;
                    }

                    // Append the action to the list if one was created.
                    if ( pAction )
                    {
                        // Set the conflict stamp in the action.
                        pAction->ConflictStamp ( dtsSourceMod );

                        // Load the body object for this record.
                        pAction->GudRecord()->LoadBody ( );

                        // Save a copy of the gud record and make sure it gets written
                        // to the temporary file for the time being.
                        TSRecord* pNewRecord = (TSRecord*) pAction->GudRecord ( )->Copy ( );
                        pNewRecord->Temporary ( TSBOOL_TRUE );

                        // Unload the body object.
                        pAction->GudRecord()->BodyObject ( NULL );

                        // Get the record from the source
                        pSource->Get ( pNewRecord );

                        // Setup the action list.
                        pAction->TempRecord ( pNewRecord );

                        pItem->SourceID ( pSource->ID ( ) );
                        pItem->CRC    ( uCRC );

                        AppendAction ( pAction );

                        // Increase the synchronization totals.
                        if ( pAction->Type ( ) == TSRECACTIONTYPE_GUD_ADD )
                            pSource->m_uAdditionsOut++;
                        else
                            pSource->m_uUpdatesOut++;
```

```
                                    // If this record was modified later than any other
                                    // new record we should indicate so in our last
                                    // category sync time.
                                    if ( dtsSourceMod > dtsLastModification && uCRC == 0 )
                                    {
                                        dtsLastModification = dtsSourceMod;
                                        pMap->LastRecordID ( pItem->SourceID ( ) );
                                    }

                                    // Save the temp record to the temporary file and
                                    // clear the memory used for it.
                                    pNewRecord->SaveBody ( );
                                    pNewRecord->BodyObject ( NULL );
                                }
                        }
                while ( pSource->MoveNext ( ) );
        }

        return iAddCount;
}


//////////////////////////////////////////////////////////////
// Generate the source delete actions.

void GetActions_SourceDeletes (
        TSSource*                       pSource,
        TSRecordMap*            pMap,
        TSDateTimeStamp&      dtsLastSync,
        TSBOOL                          bKnownDelete
        )
{
        // If the source responds to a filter for deletions then
        // get the deletions directly from them.
        if ( tsSuccess == pSource->Filter ( TSSOURCE_FILTER_DELETIONS, dtsLastSync ) )
        {
                if ( tsSuccess == pSource->MoveFirst ( ) )
                {
                        do
                        {
                                // Check to see if the record told be deleted actually
                                // exists in our record map.
                                TSRecordMapItem* pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_SOURCEID, (TSUINT32)(TSCSTR)pSource->ID ( ) );
                                if ( NULL == pItem )
                                        continue;

                                // Create the delete action and add it to the action vector.
                                AppendAction ( TSRECACTIONTYPE_GUD_DELETE, pSource, pItem );

                                pSource->m_uDeletionsOut++;

                        } while ( tsSuccess == pSource->MoveNext ( ) );
                }
        }
        else
```

```
                {
                        // Determine if there are any deletions.  If there are find them.
                        if ( TSBOOL_FALSE == bKnownDelete )
                                return;

                        // Determine all of the deletions for a given source.
                        if ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_FIRST ) )
                        {
                                do
                                {
                                        // If the record does not exist in the map, mark it for delete
                                        if ( tsSuccess != pSource->MoveTo ( pMap->CurrentMapItem()->SourceID ( ) )
                                        {
                                                AppendAction ( TSRECACTIONTYPE_GUD_DELETE,
                                                        pSource, pMap->CurrentMapItem ( ) );

                                                pSource->m_uDeletionsOut++;
                                        }
                                }
                                while ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_NEXT ) );
                        }
                }

                return;
        }


//////////////////////////////////////////////////////////////
// Generate the GUD update actions.

void GetActions_GudUpdates (
        TSSource*                pSource,
        TSRecordMap*    pMap
        )
{
        // Tell the source to stop filtering on additions/modifications
        pSource->Filter ( TSSOURCE_FILTER_CLEAR, TSDateTimeStamp() );

        // Determine if the GUD has any record for the source.
        if ( m_pStore->CurrentRecord ( TSSTORE_RECORD_FIRST ) )
        {
                do
                {
                        // Get the current record from the store.
                        TSRecord* pRecord = m_pStore->CurrentRecord ( );

                        // If the store item is not in the record map it
                        // can be marked as an add to that source.
                        TSRecordMapItem* pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_RECORDID, pRecord->UniqueID ( ) );
                        if ( NULL == pItem )
                        {
                                pItem = pMap->CreateMapItem ( NULL, pRecord );
                                AppendAction ( TSRECACTIONTYPE_CLIENT_ADD, pSource, pItem );
                        }
```

```
                              // If the item exists in the GUD, check its timestamp
                              // to the Record maps timestamp for last sync. If the
                              // the GUD record is newer we have and update
                              else
                              {
                                  // If the record was modified later than the last sync time
                                  // of the specific record then we should mark it as an update.
                                  if ( pRecord->LastModified ( ) > pItem->LastSync ( ) )
                                      AppendAction ( TSRECACTIONTYPE_CLIENT_UPDATE, pSource,
pItem );
                              }
                          }
                  while ( m_pStore->CurrentRecord ( TSSTORE_RECORD_NEXT ) );
          }

          return;
}


//////////////////////////////////////////////////////////////
// Generate the GUD delete actions.

void GetActions_GudDeletes (
          TSSource*              pSource,
          TSRecordMap*    pMap
          )
{
          // To determine whether or not there are deletions coming from the GUD we just
          // need to find all records in the record map which have the deletion flag set on
          if ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_FIRST ) )
          {
                  do
                  {
                              // If the record in the gud has been deleted, we can issue a delete
                              // to the client.
                              if ( pMap->CurrentMapItem()->Record()->Deleted ( ) == true )
                                  AppendAction ( TSRECACTIONTYPE_CLIENT_DELETE, pSource,
                                      pMap->CurrentMapItem ( ) );
                  }
                  while ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_NEXT ) );
          }

          return;
}


//////////////////////////////////////////////////////////////
// Resolve any action conflicts.

void ResolveConflicts ( )
{
          // Build the conflicts vector.
          BuildConflictsVector ( );

          // Resolve any conflicts which can automatically be done.
          ResolveAutomaticConflicts ( );
```

```
              // If there are still conflicts to resolve we must be using manual
              // resolution, therefore we need to allow the user to fixup the conflicts.
              if ( m_vecConflicts.Size ( ) > 0 )
                     DisplayDialog ( );

              // Purge actions. Run through them backwards so that the delete numbers
              // stay valid as we are deleting them.
              for ( TSNumber* pnumAction = (TSNumber*)m_vecDelActions.Last();
                     pnumAction;
                     pnumAction = (TSNumber*)m_vecDelActions.Prev ( ) )
              {
                     TSRecordAction* pAction = (TSRecordAction*)(*m_pvecActions) [ pnumAction->Value ( ) ];
                     if ( pAction == NULL )
                            continue;

                     // Delete action.
                     pAction->TempRecord ( NULL );


                     // If this type was an add then we can just delete the record map item since
                     // it isnt already in a list somewhere.
                     if ( pAction->Type ( ) == TSRECACTIONTYPE_CLIENT_ADD )
                            delete pAction->RecordMapItem ( );

                     m_pvecActions->Delete ( pnumAction->Value ( ) );
              }


              return;
}


//////////////////////////////////////////////////////////////
// Build the initial conflicts list.


void BuildConflictsVector ( )
{
              TSActionConflict* pConflict = new TSActionConflict;

              // Loop through all of the actions in the given action vector and
              // find the conflicts
              for ( TSUINT32 uAction = 0; uAction < m_pvecActions->Size(); )
              {
                     TSRecordAction* pAction = (TSRecordAction*) (*m_pvecActions)[uAction];

                     TSUINT32 uRecID = pAction->GudRecord()->UniqueID ( );

                     // Loop while the actions act on the same record.  If there is more
                     // than one action acting on the same record then we have a conflict.
                     do
                     {
                            TSRecordAction* pAction = (TSRecordAction*) (*m_pvecActions)[uAction];

                            if ( pAction->GudRecord ( )->UniqueID ( ) == uRecID )
                                   pConflict->m_vecActions.Append ( uAction );
                            else
                                   break;
```

```
                              uAction++;
                        }
                  while ( uAction < m_pvecActions->Size ( ) );

                        // If there is more than one action acting on the current record id
                        // we have a conflict.
                        if ( pConflict->m_vecActions.Size ( ) > 1 )
                        {
                              m_vecConflicts.Append ( pConflict );
                              pConflict = new TSActionConflict;
                        }
                        else
                              pConflict->m_vecActions.Clear ( );
            }

      delete pConflict;

      return;
}

//////////////////////////////////////////////////////////
// Resolve the automatic conflicts.

void ResolveAutomaticConflicts ( )
{
      TSBitField& flags = TSApplication::Config ( )->BitField ( APPCFG_GENERALFLAGS );
      TSBOOL bAutomatic = flags.Bit ( APPCFG_FLAGS_AUTOCONFLICT );

      // Iterate through all of the conflicts and resolved all which
      // can be automatically be resolved.
      for ( TSUINT32 uConflict = 0; uConflict < m_vecConflicts.Size ( );  )
      {
            TSActionConflict* pConflict = (TSActionConflict*)m_vecConflicts[uConflict];

            TSBOOL bResolved = ResolveAutomaticConflict ( pConflict, bAutomatic );

            // If the conflict was resolved, we can remove it from the list.
            if ( bResolved )
                  m_vecConflicts.Delete ( uConflict );
            else
                  uConflict++;
      }

      return;
}

//////////////////////////////////////////////////////////
// Resolve the conflict.

TSBOOL ResolveAutomaticConflict (
      TSActionConflict*         pConflict,
      TSBOOL                          bAuto
      )
{
      TSBOOL bResolved = TSBOOL_TRUE;
```

```
            // Copy the action array;
            TSNumberVector vecActionNums;
            for ( TSNumber* pnumAction = pConflict->m_vecActions.First();
                pnumAction;
                    pnumAction = pConflict->m_vecActions.Next() )
            {
                vecActionNums.Append ( pnumAction->Value ( ) );
            }

            // Step 1.  Iterate through all of the actions and resolve any conflicts between
            //      two actions acting on the same source.
            for ( TSUINT32 uAction = 0; uAction < vecActionNums.Size(); )
            {
                // Get the first action to work on.
                TSRecordAction* pAction = (TSRecordAction*)
                    ((*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction ])->Value() ]);

                // Search forward in the action vector for actions which have the same
                // source as the current action.
                TSBOOL bAdvance = TSBOOL_TRUE;
                for ( TSUINT32 uAction2 = uAction + 1;
                        uAction2 < vecActionNums.Size(); uAction2 ++ )
                {
                    // Get the first action to work on.
                    TSRecordAction* pAction2 = (TSRecordAction*)
                        ((*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction2 ])->Value() ]);

                    // If the two actions do not have the same source then continue on.
                    if( pAction2->Source ( ) != pAction->Source ( ) )
                            continue;

                    if ( pAction->ConflictStamp ( ) > pAction2->ConflictStamp ( ) )
                    {
                        m_vecDelActions.Append ( ((TSNumber*)vecActionNums[ uAction2 ])->Value
( ) );
                        vecActionNums.Delete ( uAction2 );
                    }
                    else
                    {
                        m_vecDelActions.Append ( ((TSNumber*)vecActionNums[ uAction ])->Value (
) );
                        vecActionNums.Delete ( uAction );
                        bAdvance = TSBOOL_FALSE;
                    }

                    break;
                }

                if ( bAdvance )
                    uAction++;
            }

            // Step 2/3.  Purge all client actions if there is at least one gud action.
            TSRecordAction* pFirstAction = (TSRecordAction*)
```

```
                (*m_pvecActions)[((TSNumber*)vecActionNums[0])->Value()];

        if ( TSRECACTIONTYPE_GUD_UPDATE == pFirstAction->Type ( ) ||
            TSRECACTIONTYPE_GUD_DELETE == pFirstAction->Type ( )  )
        {
            for ( TSUINT32 uAction = 0; uAction < vecActionNums.Size(); )
            {
                // Get the first action to work on.
                TSRecordAction* pAction = (TSRecordAction*)
                    (*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction ])->Value() ];

                // Once we have hit the client actions we are done with the
                // conflict resolution.
                if ( TSRECACTIONTYPE_CLIENT_DELETE == pAction->Type ( ) ||
                    TSRECACTIONTYPE_CLIENT_UPDATE == pAction->Type ( )  )
                {
                    m_vecDelActions.Append ( ((TSNumber*)vecActionNums[ uAction ])->Value()
);
                    vecActionNums.Delete ( uAction );
                }
                else
                    uAction ++;
            }

            // Step 3.  If the first action is a gud update then we can remove all
            //      gud deletes since the update always takes precedence.
            if ( TSRECACTIONTYPE_GUD_UPDATE ==
((TSRecordAction*)(*m_pvecActions)[((TSNumber*)vecActionNums[0])->Value()])->Type ( ) )
                for ( TSUINT32 uAction = 1; uAction < vecActionNums.Size ( ); )
                {
                    // Get the first action to work on.
                    TSRecordAction* pAction = (TSRecordAction*)
                        (*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction
])->Value() ];

                    // If the action is a gud delete we should purge it.
                    if ( TSRECACTIONTYPE_GUD_UPDATE != pAction->Type ( ) )
                    {
                        m_vecDelActions.Append ( ((TSNumber*)vecActionNums [ uAction
])->Value() );
                        vecActionNums.Delete ( uAction );
                    }
                    else
                        uAction ++;
                }

            // If the gud action is a delete then remove all other gud
            // actions which are deltes, we only need one.
            if ( TSRECACTIONTYPE_GUD_DELETE == pFirstAction->Type ( ) )
            {
                while ( vecActionNums.Size ( ) > 1 )
                {
                    m_vecDelActions.Append ( ((TSNumber*)vecActionNums[ 1 ])->Value() );
                    vecActionNums.Delete ( 1 );
```

```
                              }
                        }
                  else if ( vecActionNums.Size () > 1 )
                        {
                              // Find the action with the greatest modification time.  This will
                              // be the basis of our conflict merge.
                              TSUINT32 uFirstAction = 0;
                              for ( TSUINT32 uAction = 0; uAction < vecActionNums.Size(); uAction ++ )
                              {
                                    // Get the first action to work on.
                                    TSRecordAction* pAction = (TSRecordAction*)
                                          (*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction
])->Value() ];
                                    if ( pAction->ConflictStamp ( ) > pFirstAction->ConflictStamp ( ) )
                                    {
                                          pFirstAction = pAction;
                                          uFirstAction = uAction;
                                    }
                              }

                              vecActionNums.Delete ( uFirstAction );

                              // Set the first action.
                              pConflict->m_pResultingAction = pFirstAction;

                              // Change the type to a global update.
                              pFirstAction->Type ( TSRECACTIONTYPE_GLOBAL_UPDATE );

                              for ( uAction = 0; uAction < vecActionNums.Size(); )
                              {
                                    // Get the first action to work on.
                                    TSRecordAction* pAction = (TSRecordAction*)
                                          (*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction
])->Value() ];

                                    // Merge the records.
                                    TSMergeConflictVector vecConflicts;

                                    m_pAppType->SyncTypeManager()->MergeRecords (
                                          pFirstAction->TempRecord ( ), pAction->TempRecord ( ),
                                          pFirstAction->GudRecord(),
                                          pConflict->m_vecConflicts
                                          );

                                    // If we are not automatically resolving conflicts then determine whether or not
                                    // this conflict has been resolved.
                                    if ( TSBOOL_FALSE == bAuto )
                                    {
                                          if ( tsSuccess != tsMergeResult )
                                                bResolved = TSBOOL_FALSE;
                                          else if ( pConflict->m_vecConflicts.Size ( ) > 0 )
                                          {
                                                bResolved = TSBOOL_FALSE;
                                                m_bFieldConflict = TSBOOL_TRUE;
                                          }
```

```
                                       }
                                  if ( TSBOOL_TRUE == bAuto || tsSuccess == tsMergeResult )
                                  {
                                       // Delete the unnecessary action.
                                       m_vecDelActions.Append ( ((TSNumber*)vecActionNums[ uAction
])->Value() );
                                       vecActionNums.Delete ( uAction );
                                  }
                             else
                                  uAction++;
                        }
                   }
              }

         return bResolved;
}


////////////////////////////////////////////////////////////
// Perform the actions.

void PerformActions ( )
{
         // Iterate through all of the actions in the action vector and
         // perform each.  This function assumes that any conflicts in the
         // actions are already resolved.
         for ( TSRecordAction* pAction = (TSRecordAction*) m_vecActions.First ( );
              pAction;
                   pAction = (TSRecordAction*) m_vecActions.Next ( ) )
         {
                   TSApplicationSource* pAppSrc = pAction->Source()->SourceManager()->ApplicationSource( );

                   PerformAction ( pAction );
         }

         return;
}


void PerformAction ( TSRecordAction* pAction )
{
         TSRecordMapItem* pItem         = pAction->RecordMapItem ( );
         TSSource*      pSource   = pAction->Source ( );
         TSRecord*             pGudRecord = pAction->GudRecord ( );
         TSRecordMap*    pMap           = pSource->SourceManager()->RecordMap ( );

         pSource->RecordMapItem ( pItem );

         switch ( pAction->Type ( ) )
         {
                   case TSRECACTIONTYPE_CLIENT_ADD:
                        {
                             // Add the record to the source.
                             pSource->Add ( *pGudRecord );

                             TSString strID = pSource->ID ( );
```

```
                    pMap->CurrentMapItem ( TSRECORDMAP_MAP_SOURCEID,
(TSUINT32)(TSCSTR) strID );

                    // Save the clients crc for this record in the record map.
                    pItem->CRC ( pSource->CRC ( ) );

                    // Fill in the source id and add the record to the map.
                    pItem->SourceID ( strID );
                    pMap->AddMapItem ( pItem );

                    // Increment the appropriate source totals.
                    pSource->m_uAdditionsIn++;

                    // Set the last sync time of the record map item to the last
                    // modified time of the record.
                    pItem->LastSync ( pGudRecord->LastModified ( ) );

                    if ( pItem->CRC ( ) == 0 )
                            pMap->LastRecordID ( pItem->SourceID ( ) );

                    break;
                }

            case TSRECACTIONTYPE_CLIENT_UPDATE:
                {
                    // Move to the record which needs to be updated and attempt to
                    // update it.
                    if ( pItem->SourceID ( ).Length ( ) == 0              ||
                            tsSuccess != pSource->MoveTo ( pItem->SourceID ( ) )  )
                    {
                            pMap->RemoveMapItem ( pItem );
                            pAction->Type ( TSRECACTIONTYPE_CLIENT_ADD );
                            PerformAction ( pAction );
                            return;
                    }

                    pSource->Update ( *pGudRecord );

                    TSString strID = pSource->ID ( );
                    TSRecordMapItem* pFindItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_SOURCEID,

                    (TSUINT32)(TSCSTR) strID );

                    // Save the clients crc for this record in the record map.
                    pItem->CRC ( pSource->CRC ( ) );

                    // Get the source ID again, in case it changed.
                    pItem->SourceID ( strID );
                    pItem->LastSync ( pGudRecord->LastModified ( ) );

                    // Increment the appropriate source totals.
                    pSource->m_uUpdatesIn++;

                    if ( pItem->CRC ( ) == 0 )
```

```
                                      pMap->LastRecordID ( pItem->SourceID ( ) );

                        break;
                        }

        case TSRECACTIONTYPE_CLIENT_DELETE:
                        {
                        // Move to the item which needs to be deleted.
                        pSource->MoveTo ( pItem->SourceID ( ) );

                        pSource->Delete ( );

                        // Increment the appropriate source totals.
                        pSource->m_uDeletionsIn++;

                        // Delete the item from the record map.
                        pMap->DeleteMapItem ( pItem );

                        break;
                        }

        case TSRECACTIONTYPE_GUD_ADD:

                        // Load the body for the temporary record and prevent the
                        // record from being re-written to the body file by setting the
                        // memory only flag.
                        pAction->TempRecord()->LoadBody ( );
                        pAction->TempRecord()->Flags ( ).Bit ( TSRECFLAG_MEMONLY, TSBOOL_TRUE
);

                        // Copy the data from the record to the gud record.
                        pGudRecord->CopyDataFrom ( pAction->TempRecord ( ) );

                        // Get rid of the temp record
                        pAction->TempRecord ( NULL );

                        if ( tsDuplicate == m_pStore->AddRecord ( pGudRecord ) )
                                {
                                // Add to the number of records which were merged out.
                                m_iMergedRecords++;

                        TSRecord* pDupe = m_pStore->DuplicateRecord ( );

                        TSMergeConflictVector vecConflicts;
                        if ( tsSuccess != m_pAppType->SyncTypeManager()->MergeRecords (
                                        pDupe,
                                        pGudRecord,
                                        pDupe,
                                        vecConflicts ) )
                                {
                                        if ( pDupe->ConflictStamp ( ) < pAction->ConflictStamp ( ) )
                                        {
                                                pDupe->LoadBody ( );
                                                pDupe->CopyDataFrom ( pGudRecord );
                                                pDupe->ConflictStamp ( pAction->ConflictStamp ( ) );
```

```
                                        pDupe->LastModified ( TSDateTimeStamp::CurrentTime ( )
);
                                        UpdateAllSources ( pDupe );
                                }
5                           }
                        else
                        {
                                if ( pAction->ConflictStamp ( ) > pDupe->ConflictStamp ( ) )
                                        pDupe->ConflictStamp ( pAction->ConflictStamp ( ) );
10                              pDupe->LastModified ( TSDateTimeStamp::CurrentTime ( ) );
                                UpdateAllSources ( pDupe );
                        }

                        pDupe->SaveBody ( );
15                      pDupe->BodyObject ( NULL );

                        // Delete the record which was found to be a duplicate.
                        if ( tsSuccess == pSource->MoveTo ( pItem->SourceID ( ) ) )
                        {
20                              pSource->Delete ( );
                                m_vecTrashCan.Append ( pItem );
                                m_vecTrashCan.Append ( pGudRecord );
                        }
                }
25          else
                {
                        pMap->AddMapItem ( pItem );
                        pItem->LastSync ( pGudRecord->LastModified ( ) );

30                      // Set the conflict stamp for this record.
                        pGudRecord->ConflictStamp ( pAction->ConflictStamp ( ) );

                        ExpandGudAction ( pAction );
                }
35
                        // Ensure the body of the gud record is no longer loaded.
                        pGudRecord->BodyObject( NULL );

                        break;
40
                case TSRECACTIONTYPE_GLOBAL_UPDATE:
                case TSRECACTIONTYPE_GUD_UPDATE:
                {
                        // Load the body for the temporary record and prevent the
45                      // record from being re-written to the body file by setting the
                        // memory only flag.
                        pAction->TempRecord()->LoadBody ( );
                        pAction->TempRecord()->Flags ( ).Bit ( TSRECFLAG_MEMONLY, TSBOOL_TRUE
);
50
                        // Copy the data from the record to the gud record.
                        pGudRecord->CopyDataFrom ( pAction->TempRecord ( ) );
                        // Get rid of the temp record
                        pAction->TempRecord ( NULL );
55
```

```
                    if ( TSRECACTIONTYPE_GLOBAL_UPDATE != pAction->Type ( ) )
                        pItem->LastSync ( pGudRecord->LastModified ( ) );

                    // Set the conflict stamp for this record.
                    pGudRecord->ConflictStamp ( pAction->ConflictStamp ( ) );

                    ExpandGudAction ( pAction );

                    // Unload the body object
                    pGudRecord->SaveBody ( );
                    pGudRecord->BodyObject ( NULL );

                    break;
                }

            case TSRECACTIONTYPE_GUD_DELETE:

                    // Mark the GUD record as deleted.
                    pGudRecord->Deleted ( TSBOOL_TRUE );
                    pGudRecord->LastModified ( TSDateTimeStamp::CurrentTime ( ) );

                    // Set the conflict stamp for this record.
                    pGudRecord->ConflictStamp ( pAction->ConflictStamp ( ) );

                    ExpandGudAction ( pAction );

                    // Remove the item which caused the delete to occur.
                    pMap->DeleteMapItem ( pItem );

                    break;
            }
}

void ExpandGudAction (
        TSRecordAction* pAction
        )
{
        TSRECORDACTIONTYPE eType;

        // convert the original record action type to the
        // expanded type.
        switch ( pAction->Type ( ) )
        {
            case TSRECACTIONTYPE_GUD_ADD:
                eType = TSRECACTIONTYPE_CLIENT_ADD;
                break;

            case TSRECACTIONTYPE_GUD_UPDATE:
            case TSRECACTIONTYPE_GLOBAL_UPDATE:
                eType = TSRECACTIONTYPE_CLIENT_UPDATE;
                break;

            case TSRECACTIONTYPE_GUD_DELETE:
                eType = TSRECACTIONTYPE_CLIENT_DELETE;
                break;
```

```
                    }

                    // Extract the gud record to use in the following loop
                    TSRecord* pGudRecord = pAction->GudRecord ( );

                    // Issue the delete to all other clients involved in the
                    // synchronization.
                    for ( TSSource* pSource = (TSSource*) m_vecSources.First ( );
                          pSource;
                          pSource = (TSSource*) m_vecSources.Next ( ) )
                    {
                            // Dont perform any actions to this source if it is full.
                            TSApplicationSource* pAppSrc = pSource->SourceManager()->ApplicationSource ( );
                            if ( pAppSrc->Flags ( ).Bit ( SOURCE_FLAG_LOWMEMORY ) )
                                    continue;

                            if ( pSource == pAction->Source ( )            &&
                                TSRECACTIONTYPE_GLOBAL_UPDATE != pAction->Type ( )   )
                                        continue;

                            // If this record does not belong on the current source we
                            // should no consider it.
                            if ( TSBOOL_TRUE == FilterSourceRecord ( pSource, pGudRecord ) )
                                        continue;

                            TSRecordMap*    pMap  = pSource->SourceManager ( )->RecordMap ( );
                            TSRecordMapItem* pItem = pMap->CurrentMapItem ( TSRECORDMAP_MAP_RECORDID,
pGudRecord->UniqueID ( ) );

                            if ( NULL == pItem )
                            {
                                    // If the item is NULL and the action is a delete action, it
                                    // means the record is not in the source so we dont have
                                    // to delete it.
                                    if ( eType == TSRECACTIONTYPE_GUD_DELETE )
                                            continue;

                                    // Create a new map to use in the perform function.  This should
                                    // happen always if the type is ADD and could possibly happend
                                    // if the type is UPDATE and the record does not yet exist on the
                                    // destinate source.
                                    pItem = pMap->CreateMapItem ( NULL, pGudRecord );
                            }

                            // Perform the expanded action.
                            PerformAction ( &TSRecordAction ( eType,pSource,pItem ) );
                    }

                    return;
            }

            void UpdateAllSources ( TSRecord* pGudRecord )
            {
                    // Loop through all of the sources.
                    TSRecordAction Action;
```

```
        for ( TSUINT32 uSource = 0; uSource < m_vecSources.Size(); uSource++ )
        {
            TSSource*                    pSource = (TSSource*) m_vecSources [ uSource ];
            TSRecordMap*         pMap = pSource->SourceManager()->RecordMap ();
            TSRecordMapItem*      pItem = pMap->CurrentMapItem (
    TSRECORDMAP_MAP_RECORDID, pGudRecord->UniqueID ( ) );

            if ( NULL == pItem )
                continue;

            // Build the action
            Action.RecordMapItem ( pItem );
            Action.TempRecord( NULL );
            Action.Source ( pSource );
            Action.Type ( TSRECACTIONTYPE_CLIENT_UPDATE );

            // Now perform the action.
            PerformAction ( &Action );
        }

    return;
}
```

WHAT IS CLAIMED IS:

5   1. (currently amended) In a data processing environment, a method for synchronizing multiple data sets, the method comprising:

establishing a data repository for facilitating synchronization of user information maintained among more than two data sets, said data repository storing user information that is a super-set of all user information for which any user desires synchronization support;

10   storing at least one mapping which specifies how user information may be transformed for storage at a given data set;

receiving a request for synchronizing at least one data set;

based on user information stored at said at least one data set and based on said at least one mapping, propagating to the data repository from [[each of at]] said at least one

15   data set any changes made to the user information, to the extent that such changes can be reconciled with user information already present at said data repository; and

based on user information stored at said data repository and based on said at least one mapping, propagating to [[each of]] said at least one data set any changes to the user information which have been propagated to the data repository, to the extent that such

20   changes are not present at said [[each]] at least one data set.

2. (original) The method of claim 1, wherein said step of propagating to the data repository comprises:

performing selected operations of adding, updating, and deleting information at the data repository, so that the data repository reflects changes made to user information at

25   the data sets.

3.   (original) The method of claim 2, wherein said operation of deleting information comprises a logical delete operation of marking information as having been deleted.

4.   (canceled)

5.   (original) The method of claim 1, wherein said data repository and said at least one mapping comprise a grand unification database, for facilitating synchronization among multiple data sets.

6.   (original) The method of claim 5, wherein one grand unification database is created for each type of user information which is to be synchronized.

7.   (original) The method of claim 6, wherein said environment includes types of user information selected from contact, calendar, and task-oriented information.

8.   (canceled)

9.   (original) The method of claim 1, wherein each data set comprises a plurality of data records, and wherein each data record is represented within the data repository.

10.   (original) The method of claim 9, wherein each of said data records is represented within the data repository by a corresponding data record having a unique identifier.

11.   (original) The method of claim 1, wherein each mapping comprises a mapping table storing a plurality of mapping entries, each mapping entry storing at least a first identifier for indicating a particular data record in the data repository which the entry is associated with, and a second identifier for indicating a particular data record at a particular data set which is the source for the user information.

12.   (original) The method of claim 11, wherein each mapping table is associated with a particular data set.

13.   (original) The method of claim 11, wherein each mapping entry stores particular information useful for determining when its associated user information was last modified.

14.   (original) The method of claim 13, wherein said particular information comprises a last-modified time stamp, derived at least in part from the client device where the associated user information was last modified.

15.   (original) The method of claim 13, wherein said particular information comprises a checksum value, for use with a data set residing at a client device that does not support time stamps.

16.   (original) The method of claim 1, wherein said step of propagating to each of said at least one data set comprises:

performing selected operations of adding, updating, and deleting information at each of said at least one data set, so that said each reflects changes made to user information at other data sets.

17.   (original) The method of claim 16, wherein said operation of deleting information comprises physically deleting information at said each data set.

18.   (original) The method of claim 1, wherein at least one of the said data sets functions, at least in part, as said data repository.

19.   (original) The method of claim 1, wherein user information is stored at the data repository as unformatted blob data.

20.   (original) The method of claim 19, further comprising:

providing at least one type module for facilitating interpretation of user information stored as unformatted blob data at the data repository.

21. - 40.   (canceled).

DATA PROCESSING ENVIRONMENT WITH METHODS PROVIDING CONTEMPORANEOUS SYNCHRONIZATION OF TWO OR MORE CLIENTS

5

ABSTRACT OF THE DISCLOSURE

A synchronization system providing multi-client synchronization is described. By
storing the data that is actually being synchronized (i.e., storing the actual physical body of a
memo, for instance) inside an extra database, "Grand Unification Database" (GUD), (or by

10   specially-designated client data set) under control of a central or core synchronization engine,
rather than transferring such data on a point-to-point basis, the system of the present invention
provides a repository of information that is available at all times and does not require that any
other synchronization client (e.g., PIM client or hand-held device) be connected. The GUD
provides a super-set of the other client data sets. Therefore, if the user now includes an

15   additional client, such as a server computer storing user information, the synchronization system
has all the information necessary for synchronizing the new client, regardless of whether any of
the other clients are currently available. The system can, therefore, correctly propagate
information to any appropriate client without having to "go back" to (i.e., connect to) the original
client from which that data originated.